Precondition-based Modular Verification to Guarantee Data Race Freedom in Java Programs

KyungHee Kim
Intel Corp.
Hillsboro, OR 97124, USA

Beverly A. Sanders
University of Florida
Gainesville, FL 32611, USA

Neha Rungta
NASA Ames Research Center
Moffett Field, CA 94035, USA

Eric G. Mercer
Brigham Young University
Provo, UT 84602, USA

***Abstract*—We perform sound modular verification to detect data races in Java programs within a relaxed memory model. We use annotations to specify preconditions for methods in a module (a stand-alone class or a Java library) such that the preconditions guarantee data-race freedom in the module. The annotations to ensure data race freedom specify locks required and constraints on the happens-before ordering, while annotations to restrict the environment specify bounds on the number of threads and call depth. A bounded universal environment is then automatically generated. We model check the module within this environment to (a) prove that the preconditions are sufficient or (b) detect any remaining data races that do not violate preconditions. In the later case, the information of the data race is used to manually strengthen the preconditions and repeat the environment generation and model checking process until the preconditions are sufficient to guarantee data race freedom. The annotations are the contracts that the applications using the module need to know about in order to verify their own implementation. Our approach is implemented in Java RaceFinder (JRF) where the modular analysis outperforms the non-modular analysis.**

I. Introduction

Most modern computer architectures implement relaxed memory models that are not sequentially consistent (SC). Optimizing compilers may transform programs in ways that preserve the semantics of single threaded programs but violated SC for multithreaded ones. Developers are attempting to take advantage of multi-core architectures without, however, considering the impact of the underlying relaxed memory models on the correctness of their programs. Reasoning about program behaviors within a relaxed memory model is an especially hard problem. Unwanted behaviors can arise from reordering of memory accesses allowed by the relaxed memory model. These behaviors are often difficult to reproduce.

Various verification techniques have been designed to detect data races manifested on relaxed memory models [1], [2], [3], [4], [5]. In prior work, we developed a model checking technique to detect data races which allow non-SC behaviors in Java programs [4], [5]. A data race is a concurrent read or write to a shared memory location without correct ordering between threads. Given that the Java memory model (JMM) guarantees the sequential consistency of data race free programs, it is important to be able to detect data races.

In this work, we present a sound modular verification technique to ensure data race freedom in Java programs in the JMM. We have developed a set of annotations to specify preconditions for the methods in a module. The module can be either a stand-alone class or a Java library. Data race freedom is guaranteed in the module if an application program that uses the module satisfies the specified preconditions at the entry of the methods in the module. The annotations are contracts that the application using the module need to know about in order to verify their own implementation. The preconditions serve a dual purpose: (a) they are useful in defining correct usage of the methods in the module and (b) they are used to verify whether the application actually uses them correctly.

Our modular verification technique ensures that the preconditions specified by the user are sufficient to guarantee race freedom in the module. In order to ensure data race freedom, library developers use annotations that specify the locks required and constraints on the happens-before ordering. Whereas, in order to restrict the environment of the module, the developers specify annotations that bound the number of threads and call depth. Note that the use of our happens-before relation is specific to weak memory models (WMM), and the modular analysis builds on our previous work with model checking race freedom in the Java Memory Model (JMM). A bounded universal environment is automatically generated for the module. We model check the module within the universal environment to (a) prove that the preconditions are strong enough to guarantee race freedom or (b) detect any data races that do not violate the preconditions. In the later case, the conditions of the data race and information from the counterexample generated by the model checker are used to manually strengthen the preconditions. The universal environment generation and model checking process is repeated until the preconditions are sufficient to guarantee data race freedom.

The manual cost associated with annotations we believe is acceptable for WMMs because i) data races in WMMs are non-intuitive; ii) most programmers are unaware of such data races; and iii) most verification tools are equally unaware of such data races as they assume a sequentially consistent memory model which does not reflect reality. Furthermore, in our experience, defining the preconditions needed for data race freedom is not complicated. For the examples in our empirical study in this paper, the initial sets of preconditions were sufficient to show data race freedom in every benchmark test.

The modular analysis has been implemented in Java RaceFinder (JRF) and this paper includes a brief empirical study comparing the verification time to the a more standard

non-modular analysis. The contributions of this work are as follows: (a) the ability to specify preconditions required to guarantee data race freedom in the module (b) model checking the module and preconditions with an automatically generated bounded universal environment to verify whether the preconditions are indeed sufficient to guarantee data race freedom in the module (c) an empirical analysis that demonstrates a reduction in the time and space required to verify an application and module through the use of modular verification analysis.

The rest of the paper is organized as follows: Section II briefly introduces the happens-before summary used to detect data race; Section III defines the modular analysis; Section IV is the proof of correctness; Section V overviews the JPF implementation of the modular analysis; Section VI summarizes results from the empirical study over a small benchmark set; Section VII discusses other related work; and Section VIII concludes and presents future work.

II. Summarizing Happens-before

Data races in multithreaded Java programs can by detected by summarizing the happens-before relation created by the Java relaxed memory model during state space exploration in a model checker. We model the execution of a Java program as a set of memory and synchronization actions ordered by program order within each thread ($\overset{po}{\rightarrow}$) with additional orders ($\overset{so}{\rightarrow}$) from synchronizing actions between threads. Memory actions in this context include not just reading and writing memory, but interacting with locks, starting a thread, detecting thread termination, etc. Together, these two relations partially order memory actions along a single path of execution.

To understand what is read or written into memory on any given action, we define the value-written function (V) and the write-seen function (W). The value written function maps a value to a write action. The write seen function maps a write action to a read action such that for a given read action r, $V(W(r))$ returns the value read by r.

The happens-before relation ($\overset{hb}{\rightarrow}$) for the Java memory model is constructed from a relaxation of the synchronization order combined with the program order. The relaxation of the synchronization order forms a partial order called the synchronizes-with order, $\overset{sw}{\rightarrow}$. It is derived from the synchronization order according to the following rules:

- An unlock action on a monitor lock $unlock(m)$ synchronizes-with all subsequent lock actions $lock(m)$ by any thread.
- A write to a volatile variable v synchronizes-with all subsequent reads of v.
- The action of starting a thread synchronizes-with the first action of the newly started thread.
- The final action in a thread synchronizes-with an action in any other thread that detects the thread's termination such as *join* or *isAlive*.
- The writing of default values of every object field synchronizes-with the first access of any given field.

In the descriptions above, "subsequent" is determined by the synchronization order. The *happens-before* order is the partial order on the actions in an execution obtained by taking the transitive closure of the union of $\overset{sw}{\rightarrow}$ and $\overset{po}{\rightarrow}$.

We say that an execution is well-formed if it exhibits type correctness, correct behavior of locks, consistency with the sequential semantics of the program, and *happens-before consistency*. Happens-before consistency states that a read action r of variable v is allowed to see the results of a write action $w = W(r)$ if and only

$$(r \overset{hb}{\not\rightarrow} w) \wedge (\forall w' \neq w \text{ on } v, w \overset{hb}{\not\rightarrow} w' \vee w' \overset{hb}{\not\rightarrow} r)$$

The restriction simply precludes the read from returning values not yet written or stale values when a read action is ordered in the happens-before relation with any write action to the same memory location. For read and write actions to common memory locations that are not happens-before ordered, no such restriction exists, allowing for non-deterministic behavior in non-ordered read actions.

Sequentially inconsistent behavior is observed in a well-formed execution when read and write actions on a common memory location are not happens-before ordered. In particular, the write-seen function is not required to return the most recent write for an unordered read action or even the same write for unordered read actions on different threads.[1] We say that two operations *conflict* if neither is a synchronization action, both access the same memory location, and at least one is a write. A data race is a pair of conflicting actions that are not happens-before ordered.

A *sequentially consistent* (SC) execution is one where there is a total order, $\overset{sc}{\rightarrow}$, on the actions is consistent with both program order ($\overset{po}{\rightarrow}$) and synchronizing order ($\overset{so}{\rightarrow}$) and where a read r of variable v sees the results of the most recent preceding write w:

$$(w \overset{sc}{\rightarrow} r) \wedge (\forall w' \neq w \text{ on } v, w \overset{sc}{\not\rightarrow} w' \vee w' \overset{sc}{\not\rightarrow} r)$$

A Java program is *correctly synchronized* if all sequentially consistent executions are data race free; furthermore, any legal execution of a well-formed correctly synchronized program is SC [6], [7]. As such, to prove a Java program data race free, it is sufficient to only check SC executions for data races.

A multithreaded Java program can be shown free of data races by checking for races in every legal sequential execution of the program. To detect races, we summarize the happens-before order ($\overset{hb}{\rightarrow}$) during state space enumeration. Using the summary of the happens-before relation, we are able to detect unordered conflicting memory actions of a common location during model checking.

The function h summarizes the happens-before relation ($\overset{hb}{\rightarrow}$) as follows: *Addr* is the set of memory locations representing non-volatile variables in the program, *SynchAddr* is the set of memory locations representing variables with volatile semantics and locks, and *Threads* is the set of threads. The function h : *SynchAddr* $\cup$ *Threads* $\rightarrow$ 2^{Addr} maps synchronization variables and threads to sets of non-volatile variables so that

[1] We omit for brevity other causality conditions that provide safety guarantees in programs with data races [6].

a variable x such that $x \in h(t)$ means that thread t can read or write variable x without causing a data race.

In our model of Java execution, we assume that thread *main* is the single thread that initiates the program. An execution of a program P is a finite sequence of actions $a_0, a_1, \ldots, a_n$. We further define a set of static non-volatile variables $static(P)$ necessary for computing the summary function h. In the presentation, x represents a location and t represents a thread.

As the h-function definition is inductive, the base case initializes the function starting with the thread containing *main* so it only includes the static variables in the program.

$$h_0 := \lambda z.\text{if } z = main \text{ then } static(P) \text{ else } \emptyset$$

The inductive step for the h_{n+1}-function depends on the action a_n in the program execution.

The way that h_{n+1} is obtained from h_n depends on the action a_n and is computed using four auxiliary functions *release*, *acquire*, *invalidate*, and *new*. The function *release*(t, x, h) takes h and yields a new summary function by updating $h(x)$ to include the value of $h(t)$: $h \;\widehat{=}\; h[x \mapsto h(t) \cup h(x)]$. The function is used with actions by thread t that correspond to the source of a synchronizes-with ($\overset{sw}{\rightarrow}$) edge. The function *acquire*(t, x, h) takes h and yields a new function by updating $h(t)$ to include the value of $h(x)$: $h \;\widehat{=}\; h[t \mapsto h(t) \cup h(x)]$. It is used in actions that form the destination of a synchronizes-with edge. The function *invalidate*(t, x, h) removes x from $h(t')$ for all $t' \neq t$: $h \;\widehat{=}\; \lambda z.\text{if } (t = z) \text{ then } h(z) \text{ else } h(z) \backslash \{x\}$. It is used in actions where thread t writes non-volatile x. And finally, the function *new*$(t, fields, volatiles, h)$ returns a new summary function that includes the set $fields$ in $h(t)$ and initializes the previously undefined values of h for the new volatile variables:

$$h \;\widehat{=}\; \lambda z.\quad \begin{array}{l}\text{if } (t = z) \text{ then } h(t) \cup \textit{fields} \\ \text{else if } (z \in volatiles) \text{ then } \emptyset \text{ else } h(z)\end{array}$$

The function is used in actions that instantiate new objects.

The summary function h_{n+1} is inductively constructed from h_n and the next action a_n according to the rules in Table I. Data race freedom is checked on an execution at each non-volatile read action. Given a thread t and a non-volatile read action on x by t at step i of the execution, if $x \in h_i(t)$, then there is no data race on the read. We have shown that if all SC executions of a well-formed program have no races according to the h-function, then all of its legal executions are SC. JRF is our implementation of data race detection using the h-function inside of the JPF framework. It employs many optimizations to efficiently store and update the h-function [4] and is able to suggest ways to eliminate detected data races [5].

TABLE I
RULES FOR THE INDUCTIVE DEFINITION OF h_{n+1}

a_n by thread t	h_{n+1}
write a volatile field v	*release*(t, v, h_n)
read a volatile field v	*acquire*(t, v, h_n)
lock the lock variable lck	*acquire*(t, lck, h_n)
unlock the lock variable lck	*release*(t, lck, h_n)
start thread t'	*release*(t, t', h_n)
join thread t'	*acquire*(t, t', h_n)
t'.isAlive()	if $(t'.isAlive())$ then (*acquire*(t, t', h_n) else h_n
write a non-volatile field x	*invalidate*(t, x, h_n)
read a non-volatile field x	h_n
instantiate an object	*new*$(t, fields, volatiles, h_n)$

III. MODULAR VERIFICATION

In this work we provide a mechanism to library developers to specify conditions of usage that guarantee race freedom of a library module. This guarantee is provided for certain numbers of threads and certain numbers of method invocations– predefined bounds from the developer. There are four main steps that lead to the publishing of conditions of usage that guarantee race freedom. Fig. 1 represents them along the development phase and available information. The white head arrow represents the corresponding refinement as a result of a data race and condition violation. A library developer annotates each method with preconditions that ensure data race free accesses, then a general execution environment within the specified bounds is generated to close the system. Next, a combination of model checking and symbolic execution is used to check whether the preconditions are violated or not. When the preconditions are not violated and additional races are detected in the verification process, then the developer strengthens the conditions or changes the program. This process is repeated until no races are found in the library. Finally, the library can be used by an application. At this phase, the library has already been verified for the preconditions and annotated environments and the role of modular composition is to ensure these conditions are satisfied by the specific application. When any violation is discovered at this step, the only necessary refinement is to modify the usage to meet annotated conditions. While model checking the application, the model checker does not need to maintain the h-function for any internal fields of the verified library. This allows us to achieve significant savings in time and memory when checking a race in an application which heavily uses already verified libraries.

A. Annotating methods

The process begins with the developer specifying the constraints on the environment that ensure race freedom in the library. The developer annotates each method in the library module with preconditions that encode his/her design decisions regarding the data race free guarantee. The constraints that can be specified in the preconditions are: (1) a bound on the number of threads under which it is guaranteed to be race free, (2) a depth-bound that specifies the number of times a method can be safely invoked, (3) explicit locking and synchronization requirements, and (4) h-relation requirements. The rules for specifying the preconditions are as follows:

class_annotation := **thread_bound**
method_annotation := **depth_bound** (**precondition** ...)
precondition := (**condition_type** ...)
condition_type := **field in** *h* | *lock*(**field**) | *synch*(**field**)

The **thread_bound** and **depth_bound** are integer values. The class is annotated with a **thread_bound**. The thread bound

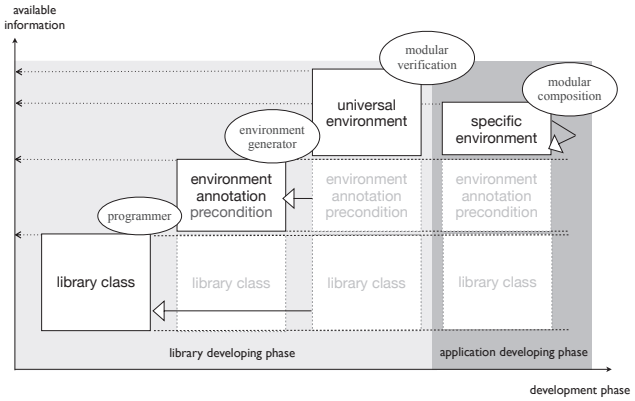


Fig. 1. The four steps in modular extension of JRF; annotating constraints and preconditions, universal environment generation, verifying the correctness of preconditions, and application composition using the library

```
1:
procedure genUnivEnv(thread_bound, depthSum, MA, Ctr)
    /* MA is the set of method and method_annotation pairs */
    /* Ctr is the set of constructors */
2:  Initialize thread t_0
3:  Non Volatile Ref := t_0.invoke(choose* Ctr)
    /* Symbolic Parameters */
4:  for each i ∈ {1, ..., thread_bound} do
5:      Initialize thread t_i
6:      for each j ∈ {1, ..., depthSum} do
7:          ⟨M, depth_bound (precondition...)⟩ := choose* MA
8:          for each precondition ∈ (precondition...) do
9:              Condition := ⟨⟩
10:             for each (condition_type...) ∈ precondition do
11:                 Condition := Condition ∘ ⟨choose* 2^(condition_type...)⟩
12:             for each i ∈ {1, ..., Condition.size()} do
13:                 C := Condition.getElementAtIndex(i)
14:                 if isLock(C) then t_i.lock(getField(C))
15:                 if isSync(C) then t_i.synchronize(getField(C))
16:             t_i.invoke(M) /* Symbolic parameters */
17:             for each i ∈ {Condition.size(), ..., 1} do
18:                 C := Condition.getElementAtIndex(i)
19:                 if isSync(C) then t_i.unsynchronize(getField(C))
20:                 if isLock(C) then t_i.unlock(getField(C))
```

Fig. 2. Algorithm to generate the universal environment

is the maximum number of threads that can concurrently access an object that is an instance of the annotated class. Each method annotation has a **depth_bound** that bounds the total number of times a method, M_i, can be invoked along a single execution path; there can be at most **depth_bound** stack frames for method, M_i, across all the different stacks in the threads of the system. The method is also annotated with a list of preconditions. The "..." in the rules is the equivalent of a Kleene star. For example, **(precondition ...)** indicates there are zero or more **precondition** elements. The preconditions are lists of **condition_type** elements. The h-relation conditions are specified with the *h*-construct, locking conditions are specified using the *lock* construct, and the conditions of being invoked within a synchronized block is specified using *synch* construct.

The semantics of the annotations are that at the method entry, all the **precondition** elements should be satisfied such that for each **precondition** at least one **condition_type** is satisfied. The elements in the list **(precondition ...)** are checked using *conjunction* while the elements in **(condition_type ...)** are checked using *disjunction*. This allows us to specify the preconditions in conjunctive normal form. Furthermore the **depth_bound** and **thread_bound** also need to be satisfied at the method entry point.

B. Universal Environment Generation

In order to check whether the specified preconditions are sufficient to actually provide race freedom, we generate a universal environment. It is the most general execution environment that satisfies the constraints specified by the preconditions. Also, it is desirable to have a universal environment that maximizes concurrent operations and minimizes any additional happens-before orderings that are not specified in the preconditions.

The algorithm to generate the universal environment is shown in Fig. 2. The algorithm can be used to generate "source code" for an executable environment. The input to the algorithm in Fig. 2 is the **thread_bound** from the annotated class; `depthSum`—the summation of all **depth_bound** values of the annotated methods in the class; MA—a set of method and annotation pairs; and Ctr—a set of constructors that are present in the annotated class. The $choose^*$ construct on lines 3, 7, and 11 is used to exhaustively explore all possible choices. For example, if the Ctr set contains two methods M_1 and M_2, the $choose^*$ generates a point of non-determinism where along one branch the constructor M_1 is invoked and along the other branch M_2 is invoked. A thread t_0 is initialized that invokes the constructor to instantiate an instance of the class and assigns the instance to a non-volatile reference. Non-volatile references ensure that additional happens-before order constraints are not added in the universal environment to keep it as general as possible.

The total number of threads initialized in the universal environment are **threads_bound** $+ 2$. The thread t_0 initializes the constructor. Finally, a $main$ thread is used to initialize all the other threads in the system. At lines 4 and 5 in Fig. 2, a thread t_i is initialized as i ranges from one to **threads_bound**.

In Fig. 2, each thread t_i goes through the inner loop at line 6 from one to the `depthSum` value in order to generate all possible method sequences in the module. The $choose^*$ construct at line 7 systematically explores all possible combinations of the methods during each iteration of the loop at line 6. We use a simple example to demonstrate the different method sequences generated by the algorithm in Fig. 2. Suppose an annotated class has methods M_1 and M_2 each with a **depth_bound** of one. The value of the `depthSum` input parameter in Fig. 2 is two (the summation of M_1 and M_2 **depth_bound** values). The following method sequences are generated: (a) $\langle M_1, M_1 \rangle$, (b) $\langle M_1, M_2 \rangle$, (c) $\langle M_2, M_1 \rangle$, and (d) $\langle M_2, M_2 \rangle$. Note that sequences (a) and (d) violate the **depth_bound** precondition and are handled in the next verification step. Each method invocation is added at line

16. All methods (including constructors) are invoked with symbolic parameters.

The algorithm adds locking and synchronization conditions at lines 14 and 15 before invoking the method at line 16 and then releases the constraints at lines 19 and 20 in Fig. 2. The algorithm first iterates through the **(precondition ...)** list of the method and for each **precondition** explores all possible choices in the power set of **condition_type** elements. Recall that the semantics state only one of the **condition_type** needs to be satisfied, but the universal environment needs to check the satisfiability of all possible combinations of the elements in **(condition_type ...)**. The algorithm constructs an ordered list $Condition$. It uses the $\circ$ operator to append elements in the set picked by the $choose^*$ construct to the $Condition$ list. We then iterate over the list of $Condition$ elements in order to add the locking and synchronization conditions before invoking the method (lines $14-16$). Then in the reverse order of acquisition the lock and synchronization blocks are released (lines $19-20$). The *h* precondition is not a programmatic precondition but rather a mechanism to track data races during verification, hence, it is handled in the verification step.

Invoking the combinations of various method sequences in the environment and using symbolic complex data is similar to the approach presented in [8]. In this work, however, we combine the method sequence generation with multiple threads for the verification of data-race freedom rather than unit test case generation for code coverage in sequential methods. We believe this combination to be novel.

C. Verification of the race freedom of the module

The annotated library or module is then verified within the generated universal environment to check whether the preconditions are sufficient to guarantee race freedom or not, and if races do exist under the preconditions, use the discovered races to strengthen the preconditions or modify the code. The library along with the universal environment has a set of threads $\{main, t_0, t_1, \ldots, t_{\textbf{thread\_bound}}\}$ where each t_i is a thread with a unique identifier $id \rightarrow \{0, 1, \ldots, \textbf{thread\_bound}+1\}$; while $\mathcal{V}_{sym}$ is a finite set of symbolic variables declared in the universal environment.

The verification is performed under a runtime environment that implements an interleaving semantics over the threads in the program. The runtime environment operates on a program state s that contains: (1) valuations of any concrete variables in the library; (2) for each thread, t_i, values of its local variables, runtime stack, locks acquired or waiting to be acquired, its current program location, and an h relation; (3) the symbolic representations and values of the variables in $\mathcal{V}_{sym}$; and (4) a path constraint, ϕ, (a set of constraints) over the variables in $\mathcal{V}_{sym}$. We present below some helper functions on state s in order to define the execution semantics of the system:

- `getLoc`(s, i) returns the current program location of the thread that has the identifier i in state s.
- `getEnabledThreads`(s) returns a set of identifiers of the threads enabled in s. A thread is *enabled* if it is not blocked—waiting to acquire a lock or waiting to join.

Given a program state, s, the runtime environment generates a set of successor states, $\{s_0, s_1, \ldots, s_n\}$ based on the following rules $\forall i \in \texttt{getEnabledThreads}(s) \wedge l := \texttt{getLoc}(s, i)$:

1) If l is a conditional branch with symbolic primitive data types in the branch predicate, P, the runtime environment can generate at most two possible successor states. It can assign values to variables in D_{sym} to satisfy the path constraint $\phi \wedge P$ for the target of the true branch or satisfy its negation $\phi \wedge \neg P$ for the target of the false branch.
2) If l accesses an object of type T, then the runtime environment generates the successor states where the object is initialized to: (a) null, (b) references to new objects of type T and all its subtypes, and (c) existing references to objects of type T and all its subtypes [9].
3) If neither rule 1 nor rule 2 is applicable, then the runtime environment generates a single successor state by executing program location, l, in thread t_i.

The rules specified above systematically explore non-determinism arising from different thread choices as well as the choices arising from operations on symbolic data.

A depth-first search is used to systematically generate and search the reachable state space generated by the runtime environment. Note that the runtime environment can only check the satisfiability of path constraints that are decidable and solvable by a particular constraint solver. Also, to limit the possibility of an infinite search space resulting from symbolically executing programs with loops, an additional depth bound for the symbolic execution is provided.

During verification, the $lock$ or $sync$ preconditions are never violated since the universal environment invokes the method only after acquiring the locks and synchronization elements, as shown in Fig. 2. Recall, however, that the **depth_bound** can potentially be violated along some method sequences. In order to handle that scenario, execution along a path is only explored up to the point where the **depth_bound** precondition is violated. When the **depth_bound** precondition is violated, the program state is treated as an end of the path and the search backtracks.

When an *h* precondition (**field in** *h*) violation is detected, the h function of the currently executing thread will be expanded. Recall that the universal environment generation process does not take into account the *h* preconditions since the h function is a verification artifact rather than a programmatic construct. The h relation is updated by adding the **field** (that is violated). When a (**field in** *h*) precondition is violated for a thread, t_i, it means that another thread, t_j, contains **field** in its $h(t_j)$ function. All variables updated by t_j before the update to **field** are also added to the h function of t_i to simulate the happens-before order from the last write to **field** to it's current access.

At the end of model checking, the preconditions are correct when no races are detected or the existing races are determined to be benign by the module developer. Data race freedom of a module means that the internal fields and methods of the module are free from a data race for any concrete environment that satisfies the preconditions and runs within the specified

bounds. If, however, a race is detected during model checking, then it demonstrates that the preconditions are not strong enough to ensure race freedom. At this point the developer has two choices, either (1) strengthen the preconditions or (2) modify the code and repeat the process of generating the universal environment and verifying the library within the environment until no races exist.

D. Verification of the Application

The application along with the verified module is checked for data races exploiting the verification results of the module. When a verified library is used in an application, the internal non-volatiles are not maintained in the h function. Instead, the preconditions at every library method invocation are checked; a precondition violation report indicates inconsistent use of a library method. The savings in the modular verification are obtained by the fact that the libraries' internal fields do not need to be maintained in the h function when verifying the composed system.

We can conclude that the application is free from data races when no precondition violations are reported. Furthermore, if no races are reported on non-volatile fields defined outside the library module, the entire system is guaranteed to be sequentially consistent without checking library internal non-volatiles.

IV. Theoretical Results

In this section, we justify the soundness of modular race checking with respect to the race free guarantee.

Definition 1: Given two well-formed executions, $E_1 = \langle A_1, P_1, \overset{po}{\rightarrow}_1, \overset{so}{\rightarrow}_1, W_1, V_1\rangle$ and $E_2 = \langle A_2, P_2, \overset{po}{\rightarrow}_2, \overset{so}{\rightarrow}_2, W_2, V_2\rangle$, we say that E_1 and E_2 are equivalent with respect to set of actions A, denoted $E_1 =_A E_2$, if and only if $A \subseteq A_1 \cap A_2$, $\overset{po}{\rightarrow}_1 |_A = \overset{po}{\rightarrow}_2 |_A$, and $\overset{so}{\rightarrow}_1 |_A = \overset{so}{\rightarrow}_2 |_A$, where $|_A$ denotes projection onto the set A.

Suppose L is a library with internal non-volatiles $fields_L$ that is only accessible through the methods of L. Let us assume a universal environment U with bounding constraint B and any arbitrary concrete execution context C of L satisfying the bounding constraint B with $E^U = \langle A^U, P^U, \overset{po}{\rightarrow}^U, \overset{so}{\rightarrow}^U, W^U, V^U\rangle$ and $E^C = \langle A^C, P^C, \overset{po}{\rightarrow}^C, \overset{so}{\rightarrow}^C, W^C, V^C\rangle$ denoting arbitrary sequentially consistent executions of U and C, respectively. When all such E^C satisfy the preconditions P of L in bound B and all E^U are race free on $fields_L$, then all, E^C are race free on $fields_L$ in the given bound B. This assumes that the symbolic execution engine has no restriction. The following two lemmas will justify this.

Lemma 1: For an arbitrary E^C, there exists a maximal prefix of E^C, denoted E^C_n when its length is n, with an equivalent execution m prefix of E^U, denoted E^U_m, w.r.t. A_L, where A_L is the set of actions in L and the actions satisfying the *lock* and *synch* preconditions of L. Such a maximal prefix of E^C, E^C_n, is defined as the longest prefix of E^C s.t. E^C_{n+1} has no equivalent execution in any prefix of E^U without symbolic restriction.

Proof: The proof is by contradiction. Let us assume that an execution $E^{C\prime}$ has no such maximal prefix that has an equivalent execution in the set of any prefixes of E^U with satisfying B w.r.t. A_L. Given that $E^{C\prime}$ satisfies all preconditions and constraints of L, we can construct an execution $E^{U\prime}$ corresponding to a path of the environment U by choosing the same actions as $E^{C\prime}|_{A_L}$ at each transition. Moreover, given that the parameters are represented symbolically, we can choose the same W and V for $E^{U\prime}$ as $E^{C\prime}$, i.e., $E^{U\prime} = \langle A^{C\prime}|_{A_L}, P^U, \overset{po}{\rightarrow}^{C\prime}|_{A_L}, \overset{so}{\rightarrow}^{C\prime}|_{A_L}, W^{C\prime}|_{A_L}, V^{C\prime}|_{A_L}\rangle$. Such a transition choice is always available in a universal environment because $E^{C\prime}$ satisfies the bounding constraint B and there are enough transition choices in U to cover all different interleavings within B. The verification rule in section III-C guarantees that this path is not ignored since this path satisfies all *lock* and *synch* preconditions and bounding constraints B. When such a choice of symbolic representation is restricted at the m^{th} action in $E^{U\prime}$, we can choose an n prefix of $E^{C\prime}$ where $A^{C\prime}_n|_{A_L} = A^{U\prime}_m|_{A_L}$. This contradicts the assumption. ■

Lemma 2 shows that the h that includes the non-volatile fields in a library is minimal in the universal environment. In other words, when a non-volatile field is in the h of a current thread at some execution step in the universal environment, it is guaranteed to be in the h of a current thread at the equivalent execution step in any equivalent concrete environment.

Lemma 2: Suppose h^{-1} is the inverse of h where $h^{-1}(x)$ is the set of memory locations $v \in$ (*SynchAddr* $\cup$ *Threads*) such that $x \in h(v)$. When h^U and h^C denote h for two equivalent executions, E^U and E^C w.r.t. A_L, respectively, $\forall x \in fields_L$, $(h^U)^{-1}(x) \subseteq (h^C)^{-1}(x)$ holds for all prefixes of $E^U|_{A_L}$.

Proof: The proof is by induction on the length of the prefix of $E^U|_{A_L}$.

Basis. We have a length 0 prefix of $E^U|_{A_L}$. Since no action in A_L happens, $\forall x \in fields_L$, $(h^U)^{-1}(x) = (h^C)^{-1}(x) = \phi$.

Inductive Step. Assume $\forall x \in fields_L$, $(h^U)^{-1}(x) \subseteq (h^C)^{-1}(x)$ holds for $(E^U|_{A_L})_n$. We will show that it also holds for $(E^U|_{A_L})_{n+1}$ for all possible $(n+1)^{th}$ action types.

1) When the $(n+1)^{th}$ action is an action satisfying the lock and synch preconditions of L, the $(n+1)^{th}$ action is either *release* or *acquire* and $(h^U)^{-1}(x) \subseteq (h^C)^{-1}(x)$ is preserved by the h update rule in Table I.
2) When the $(n+1)^{th}$ action is either *release* or *acquire* using volatile write or read, *invalidate*, or h irrelevant actions, $(h^U)^{-1}(x) \subseteq (h^C)^{-1}(x)$ is preserved by the h update rule in Table I.
3) Otherwise, the $(n+1)^{th}$ action is either an instantiation or a publication of the L object or an invocation of a method in L.
   a) When the action is an instantiation of the L object, it will add the instantiating thread to both $(h^U)^{-1}(x)$ and $(h^C)^{-1}(x)$ for all x in $fields_L$. This preserves $(h^U)^{-1}(x) \subseteq (h^C)^{-1}(x)$.
   b) When the action is a publication, given that the reference in U is defined as non-volatile, this publi-

cation will not change $(h^U)^{-1}(x)$. If the reference in C is a volatile, this will add the current thread into $(h^C)^{-1}(x)$. Otherwise, $(h^C)^{-1}(x)$ remains the same. This preserves $(h^U)^{-1}(x) \subseteq (h^C)^{-1}(x)$.

c) At a method invocation, (1) When h precondition is not violated in U, $(h^U)^{-1}(x)|_{n+1} = (h^U)^{-1}(x)|_n$. (2) When the h precondition is violated in U, this will add the violated field, f, and all other memory locations Y written prior to that by the same thread $t\prime$ to the h of current thread t. Since the memory locations in Y were last updated by $t\prime$, $Y \subseteq h(t\prime)$. $(h^U)^{-1}(f)|_{n+1} \leftarrow (h^U)^{-1}(f)|_n \cup \{t\}$ and $(h^U)^{-1}(y)|_{n+1} \leftarrow (h^U)^{-1}(y)|_n \cup \{t\}$ for all $y \in Y$. The assumption guarantees that the h precondition is satisfied in C and $\{t\} \subseteq (h^C)^{-1}(f)|_{n+1}$. If $t\prime = t$, $\{t\} \subseteq (h^C)^{-1}(y)|_{n+1}$ for all $y \in Y$. When $t\prime \neq t$, f has been added to t after the last write of f through *acquire* of v by t preceded by the *release* of v by $t\prime$. At the time of *release* of v by $t\prime$, $\{f\} \cup Y$ had been added to $h(v)$ since $\{f\} \cup Y$ had been in $h(t\prime)$. When t acquires v, it gets added $\{f\} \cup Y$ to $h(t)$. This concludes that $\{t\} \subseteq (h^C)^{-1}(f)|_{n+1}$ and $\{t\} \subseteq (h^C)^{-1}(y)|_{n+1}$ for all y in Y.
In all cases, $(h^U)^{-1}(x) \subseteq (h^C)^{-1}(x)$ is preserved. ■

Theorem 1 justifies the preconditions that were verified in U can guarantee the data race freedom on internal fields of L in the maximal set of actions of C for which they are equivalent.

Theorem 1: When a set of preconditions are verified to be correct in a universal environment U with bounding constraint B, any arbitrary concrete environment C within B is guaranteed to be free from data races on any internal fields of L up to the maximal set of actions in E^C that is equivalent to the prefixes of E^U without symbolic execution restriction, if all preconditions of L are satisfied in all sequentially consistent executions E^C of C.

Proof: The proof follows immediately from lemma 2. Since $\forall x \in fields_L$, $(h^U)^{-1}(x) \subseteq (h^C)^{-1}(x)$ holds for all prefixes of $E^U|_{A_L}$, $x \in h^U(t)$ at $(E^U|_{A_L})_n$ guarantees $x \in h^C(t)$ at $(E^C|_{A_L})_n$. ■

When symbolic execution can cover all paths of a module (the module does not contain any loops, recursion, and generates constraints that can be solved by the constraints solver) we can strengthen Theorem 1 to guarantee the data race freedom for all of E^C. Since C is proved to have no race on $fields_L$, L can be trusted and safely excluded from h without hurting the soundness of JRF in C.

V. Extending JPF

The modular verification approach is implemented within the Java Pathfinder (JPF) tool kit [10]. JPF is an explicit state model checker for Java bytecode. It systematically explores thread non-determinism. We use the Java RaceFinder (JRF), [4], [5], and Symbolic Pathfinder (SPF) extensions [11], [12]. JRF incorporates knowledge of the JMM while SPF is a symbolic execution engine used to track symbolic variables while verifying the module within the universal environment. The modular analysis extends the non-modular analysis of JRF.

A. The Listener Implementation

JPF supports a Listener interface that can be used to extend its functionality. The interface notifies low level events at the JPF Java virtual machine level through preregistered callback functions. Types of these events are VM related events, such as *instructionExecuted*, *threadStarted*, and *objectLocked*, search related events, such as *searchAdvanced*, *searchBacktracked* and *propertyViolated*; those events are defined in `VMListener` and `SearchListener` respectively. JRF listener inherits `PropertyListenerAdapter`, which implements both `VMListener` and `SearchListener` interfaces. Callback functions inherited from `SearchListener` manage the stack structure to store Δ of h, and callbacks from `VMListener` manage h, as described in Section II. The operations *acquire*, *release*, *invalidate* and asserting *norace* are performed as appropriate when execution of memory model related instructions occur. To facilitate the selective instrumentation of instructions, JPF also provides a visitor pattern `InstructionVisitor` in the `gov.nasa.jpf.jvm.bytecode` package and JRF visitor used this feature to implement optimized representation of h including lazy representation of array elements [13], [4], [5]. JPF *Field Factory* feature enables JRF to intercept all accesses to the fields including the accesses originated from MJI (Module Java Interface) codes. JRF maintains efficient h representation using the visitor and field factory extensions.

The modular extension to JRF adds another `InstructionVisitor` to check preconditions in both verification (Section III-C) and composing phases (Section III-D) at every method entry (*invoke* instructions).[2]

B. Saving Constraints as Attributes

The environment constraints and preconditions are annotated using the `java.lang.annotation` package and `gov.nasa.jpf.jvm.AnnotationInfo` class. The annotated information is saved as an attribute of an object using the attribute system in JPF. An attribute is a storage extension to save additional values for local variables, fields, and objects and JPF maintains its reference while searching the state space as it restores the values of fields and objects upon backtracking. Since the environment constraints are additional data to be maintained in JRF modular extension and JPF only restores the attribute reference, any update of an environment constraint requires copy-on-write to the attribute of the corresponding object.

C. ChoiceGenerator and SPF for Universal Environment

Though the thread interleaving is the dominating source of search space, it is also necessary to consider the nondeterministic data in model checking. JPF provides two different choices, scheduling choice and data choice. Scheduling

[2] `BytecodeFactory` could be used to implement the same functionality but was not an option in this case since SPF had already used this feature and JPF did not support multiple bytecode factories.

choices related to the h abstraction in partial order reduction is implemented in the JRF and various search algorithms and optimization approaches are discussed in [4], [5]. In this modular extension, the universal environment explicitly enumerates choice for orders of method invocation and relies on SPF to define possible parameter values for each method invocation.

VI. Experimental Results

In this section, we first demonstrate an experience using a simple example. Consider the slightly modified `UnboundedQueue` Java library from [14], shown in Fig. 3. In `UnboundedQueue`, the non-volatile shared fields `head` and `tail` are protected by explicit locks `deqLock` and `enqLock`, respectively. However, the `size()` method does not lock the fields and requires the user to acquire both locks before invoking `size()`. This requirement is added as a comment in the code, as shown in Fig. 3. It is possible that the application programmer is not aware of this requirement and does not read the comments in the library he is using. In other cases, the comment may be ambiguous and it might be hard for the application programmer to determine the library developer's true intent.

```
public class UnboundedQueue {
   private static final int EMPTY= Integer.MIN_VALUE;
   public final ReentrantLock enqLock;
   public final ReentrantLock deqLock;
   Node head, tail;

   public UnboundedQueue() {
      enqLock = new ReentrantLock();
      deqLock = new ReentrantLock()
      head = new Node(EMPTY);
      tail = head;
   }
   public int deq() throws EmptyException {
      int result;
      deqLock.lock();
      try {
         if (head.next == null)
            throw new EmptyException();
         result = head.next.value;
         head = head.next;
      } finally { deqLock.unlock(); }
      return result;
   }
   public void enq(int x) {
      if (x == EMPTY)
         throw new NullPointerException();
      enqLock.lock();
      try {
         Node e = new Node(x);
         tail.next = e;
         tail = e;
      } finally { enqLock.unlock(); }
   }
   public int size() { /*requires:enqLock, deqLock*/
      int i=(head==tail?0:1);
      for (Node tmp=head.next; tmp!=null &&
                  tmp!=tail ; tmp=tmp.next, ++i);
      return i;
   }
   protected class Node {
      final int value;
      volatile Node next;
      Node(int x) { value = x; next = null; }
   }
}
```

Fig. 3. `UnboundedQueue` library

```
@class (threads_bound=3)
public class UnboundedQueue {
   . . .

   public UnboundedQueue() { . . . }

   @method (depth_bound=2)
   @precondition (h="CURRENT_THREAD WITH THIS")
   public int deq() throws EmptyException { . . . }

   @method (depth_bound=2)
   @precondition (h="CURRENT_THREAD WITH THIS")
   public void enq(int x) { . . . }

   @method (depth_bound=2)
   @precondition (h="CURRENT_THREAD WITH THIS",
                  lock={"enqLock"}, lock={"deqLock"} )
   public int size() { . . . }
   . . .
}
```

Fig. 4. `UnboundedQueue` library with precondition annotation

The library developer annotates the `UnboundedQueue` library with precondition $lock =$ "$enqLock$" and "$deqLock$" at `size()` to add the locking constraint in Fig. 4. The bounds on the threads and call depth are also shown. The constructor can only be invoked once along a given path. The h precondition contains the "this" object that specifies the object should be safely published. [3]

The bounded universal environment generated for the `UnboundedQueue` example with the preconditions of Fig. 4 is shown in Fig. 5. The environment is for the most part a Java program. The one element specific to JPF is the `Verify.getInt(min, max)` calls. This construct creates a point of non-determinism. JPF creates `max-min` choices. For example the line of code `int c = gov.nasa.jpf.jvm.Verify.getInt(1,3)` in Fig. 5 generates three choices where the value of `c` is one, two, and three respectively. The universal environment is used to verify that the preconditions for the specified bound to guarantee data race freedom.

Let's assume an application `FairMessage` which is slightly modified from the junit test driver for `UnboundedQueue` in [14] to include a call to the other library `DisBarrier`. In `FairMessage`, two threads calls `enq` and two threads calls `deq` followed by a barrier `await` before next iteration. The main thread prints the size of the queue after starting the `EnqThread` and `DeqThread` workers without any synchronization. This is the source of a data race on `head` and `tail` at the `queue.size()`. JRF reported races on `head` and `tail` at the first two lines of `size()` in `UnboundedQueue.java` and JRF-E suggested to change these fields to volatile or to lock before accessing them in `UnboundedQueue` rather than `FairMessage` as shown in Fig. 7. On the other hand, the modular race analysis presented in this paper reported the precondition violation at `System.out.println("queue size = "+queue.size());` in `FairMessage.java` as Fig. 8.

[3] An object is *published* when it reference is made visible to other threads. Unsafe publication (Section 3.5 of [15]) is a common error that can allow an object to become visible before its initialization is complete.

```
public class UnboundedQueueVerify {
   UnboundedQueue obj;
   @Symbolic("true")
   int sym0;

   public static void main(String[] args) {
      new UnboundedQueueVerify().doTest();
   }
   void doTest() {
      for (int i=0 ; i < 1 ; ++i) new Group1Thread().start();
      for (int i=0 ; i < 3 ; ++i) new Group2Thread().start();
   }
   class Group1Thread extends Thread {
      public void run() {
         for ( int i=0 ; i < 1 ; ++i) {
            int c = gov.nasa.jpf.jvm.Verify.getInt(1,1);
            if ( c == 1 ) obj = new UnboundedQueue();
         }
      }
   }
   class Group2Thread extends Thread {
      public void run() {
         for ( int i=0 ; i < 6 ; ++i) {
            while ( obj==null);
            int c = gov.nasa.jpf.jvm.Verify.getInt(1,3);
            if ( c == 1 ) {
               obj.deqLock.lock();
               obj.enqLock.lock();
               try { obj.size(); }
               finally{
                  obj.enqLock.unlock();
                  obj.deqLock.unlock();
               }
            }
            else if ( c == 2 ) {obj.enq(sym0);}
            else if ( c == 3 ) {
               try{obj.deq();} catch (EmptyException e) {}
            }
         }
      }
   }
}
```

Fig. 5. Generated universal environment for `UnboundedQueue`

```
public class FairMessage {
   UnboundedQueue queue = new UnboundedQueue();
   DisBarrier bar = new DisBarrier(NUM_THREAD);
   static final int NUM_THREAD=2, PER_THREAD=2;

   public static void main(String[] args) {
       (new FairMessage()).run();
   }
   private void run() {
      for ( int i=0 ; i < NUM_THREAD ; ++i)
      { new EnqThread(i).start();
        new DeqThread().start();
      }
      System.out.println("queue size = "+queue.size());
   }
   class EnqThread extends Thread {
      int  id;
      EnqThread(int i) { id = i; }
      public void run() {
         for (int i = 0; i < PER_THREAD; i++) queue.enq(id+i);
      }
   }
   class DeqThread extends Thread {
      public void run() {
         for (int i = 0; i < PER_THREAD; i++)
            try {
              queue.deq();
              bar.await();
            } catch (EmptyException ex) {}
      }
   }
}
```

Fig. 6. `FairMessage` uses `UnboundedQueue` and `DisBarrier`

```
================================================
JRF results
================================================ data race #1
edu.ufl.cise.jrf.util.HBDataRaceException
         at THREAD      (java.lang.Thread@ from null)
         to MEMORY      (jrfm.UnboundedQueue@.tail
                          from "volatile UnboundedQueue queue = new UnboundedQueue();"
                          at jrfm/FairMessage.java:10 in (<init>))
         in INSTRUCTION (getfield)
         of SOURCE      ("for (Node tmp=head.next; tmp!=null &&
                                    tmp!=tail ; tmp=tmp.next, ++i);"
                          at jrfm/UnboundedQueue.java:72)
. . .
================================================
JRF-E results
_____________________________________________ analyze counter example
data race source statement : "putfield" at jrfm/UnboundedQueue.java:58  :
          "tail = e;"
          by thread 1
data race manifest statement : "getfield" at jrfm/UnboundedQueue.java:72:
          "for (Node tmp=head.next; tmp!=null &&
                      tmp!=tail ; tmp=tmp.next, ++i);"
          by thread 0

Change the field "jrfm.UnboundedQueue@.tail
         from "volatile UnboundedQueue queue = new UnboundedQueue();"
         at jrfm/FairMessage.java:10 in (<init>)" to volatile.

    Lock "java.util.concurrent.locks.ReentrantLock@
           from "enqLock = new ReentrantLock();"
           at jrfm/UnboundedQueue.java:25 in (<init>)"
           before accessing (jrfm.UnboundedQueue@.tail)
. . .
```

Fig. 7. Race in `FairMessage` detected by JRF and suggestions provided by JRF-E

```
================================================
JRFM-ComposeModule results
================================================ precondition violation #0
in "jrfm.UnboundedQueue.size()"
    the lock precondition of method (size) "enqLock, deqLock" is violated.
     at "System.out.println("queue size = "+queue.size());"
     in "jrfm.FairMessage.run(FairMessage.java:23)"
================================================ precondition violation #1
in "jrfm.UnboundedQueue.size()"
    the lock precondition of method (size) "enqLock, deqLock" is violated.
     at "assert (queue.size() == 0);"
     in "jrfm.FairMessage.run(FairMessage.java:17)"
. . .
```

Fig. 8. Precondition violations in `FairMessage` detected by JRF modular extension

The important difference in the results of `UnboundedQueue` example is the target of verification. JRF and its other extensions are focusing on the verification of the whole target application, on the other hand, the modular method presented in this paper partitions the target into trusted libraries which had already verified for the predefined consistent usage pattern and untrusted modules which should meet those constraints. The data race in the application should be eliminated by modifying the untrusted codes which violates the preconditions of immutable libraries.

The rest of this section will present the experimental results using the JRF modular extension for the selected set of test cases used in JRF[4]. Note that JRF is one step verification for individual application context and modular approach consist of a library verification per each library and a constraint checking per individual application contexts. We can assume that a library is verified once at the time of its development phase and referenced multiple times in different application contexts so that the one-time library verification overhead is acceptable. This is discussed further in Fig. 9.

Table II summarizes the resources consumed for the library verification. LOC specifies the lines of code in the library and LOP are the precondition annotations in lines (one precondition per line). The time and memory expended in the environment generation (env. generation) and the resources

TABLE II
TIME AND MEMORY CONSUMED IN THE LIBRARY VERIFICATION STEP

library	annotation		env. generation		precondition verification			
	LOC	LOP	time	memory	jpf-states	h-states	time	memory
Peterson	80	13	0.45 sec	1292 KB	3504	2919	193 sec	864 MB
Bakery	129	15	0.48 sec	1292 KB	1316	3278	172 sec	966 MB
CountDownLatch	108	5	0.40 sec	1292 KB	1221	2451	405 sec	864MB
UnboundedQueue	89	7	0.46 sec	1292 KB	2005	2141	651 sec	1168 MB
DisBarrier	92	3	0.41 sec	1292 KB	1353	10592	1310 sec	952 MB
ConcurrentStack	112	5	0.46 sec	1292 KB	3060	1933	159 sec	973 MB
llConcurrentStack	194	5	0.48 sec	1292 KB	12631	26775	3687 sec	2030MB
llConcurrentStack_v2	194	5	0.48 sec	1292 KB	12043	25586	3320 sec	1966 MB
ConcurrentLinkedQueue	119	5	0.47 sec	1292 KB	2896	27915	1899 sec	984 MB

TABLE III
TIME TAKEN AND MEMORY USED IN MODULAR VS. NON-MODULAR VERIFICATION

example	LOC	configuration	jpf-states	h-states	time	memory
Peterson	194	modular	344	40	32 sec	863 MB
		JRF	344	133	35 sec	863 MB
Bakery	258	modular	7410	1070	819 sec	1663 MB
		JRF	7504	24152	4067 sec	1825 MB
CountDownLatch	211	modular	509	301	147 sec	834 MB
		JRF	509	622	202 sec	834 MB
UnboundedQueue	165	modular	384	50	43 sec	857 MB
		JRF	384	603	106 sec	860 MB
DisBarrier	188	modular	1443	1021	191 sec	984 MB
		JRF	1443	1500	280 sec	983 MB
FairMessage*	280	modular	1648	1090	314 sec	992 MB
		JRF	1648	6183	1357 sec	1017 MB
ConcurrentStack	427	modular	391	13	29 sec	717 MB
		JRF	391	397	48 sec	862 MB
llConcurrentStack	447	modular	1623	2191	530 sec	859 MB
		JRF	1623	2515	551 sec	947 MB
llConcurrentStack_v2	447	modular	1535	2082	495 sec	602 MB
		JRF	1535	2407	523 sec	603 MB
ConcurrentLinkedQueue	349	modular	170	34	43 sec	483 MB
		JRF	170	143	66 sec	483 MB

* `FairMessage` uses two libraries `UnboundedQueue` and `DisBarrier`.

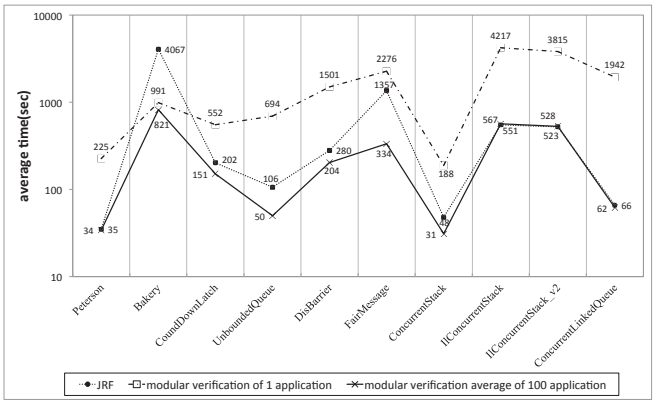
expended in the verification process are summarized. The h annotation in the method contain "this", the thread bounds are set to three, the call depth bounds are set to three in `DisBarrier` and two in all others, and the `UnboundedQueue` contains locking preconditions.

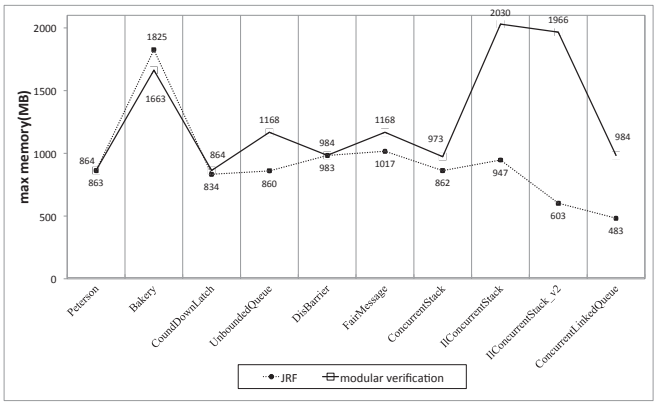
While considering the cost incurred during precondition verification phase, it is important to consider the following points: i) the verification overhead is amortized over each use of the library because the manual specification and verification of the conditions are done only once by a library developer; ii) the conditions ensure the library is used in a data race free way in any deployment context (useful for adopters); and iii) applications that employ several libraries are easier to verify for race freedom as each annotated library reduces the verification burden.

Table III presents the experimental results comparing JRF and modular analysis of applications that use the verified preconditions of the module. Note that the a jpf-state encompasses the status of heap and threads such as values of fields, program counters, and the status of threads, while a h-state considers the h-relation in addition to the jpf-state. The modular analysis explores fewer states than JRF as expected mostly due to the savings in h-states. The modular analysis utilizes less time and memory compared to JRF in our examples. The `FairMessage` example saves more time and memory than `UnboundedQueue` and `DisBarrier` since it uses both libraries and the savings are compositional. The library modules that use very few non-volatile fields and the implementation does not have many *release-acquire* operations, the overhead of checking preconditions may degrade performance. Those libraries we believe are not very suitable for the modular analysis presented in this work.

Fig. 9 represents the temporal and spatial requirements of JRF and the modular approach. Based on the assumption that a library verification is necessary only once and its result can be applied to multiple contexts, the time for the modular approach in Fig. 9(a) is taken by averaging (*environment generation time+precondition verification time+sum of composition times for application contexts*)/(*total number of application contexts*). When considering only one application context, the precondition verification for universal environment overhead outweigh the gain of modularization.[4] However, this overhead is paid back by increasing number of library uses in different contexts. As given in Fig. 9(a), 100 uses of the verified library compensates for the verification overhead in time. Memory requirements are not accumulative and only maximum memory amount required in each verification step is important. Fig. 9(b) shows that the universal environment, which covers more space than individual application contexts,

[4] `Bakery` was one exception where the application context used more than one instance of the library class.





(a) Comparison of times spent in JRF for the application given in Table III; JRF execution time without this modular extension, the sum of verification and composition time for the libraries given in Table II and application contexts in Table III, and average times when we assume the library in Table II is used by 100 different application contexts with average composition phase time as Table III are compared.

(b) Max memory required by JRF and modular verification; Note that modular verification data represents the maximum of the three memory requirements in environment generation, precondition verification, and modular composition.

Fig. 9. Comparison of JRF and modular extension resource requirements

is the most space consuming phase. Though, memory is not the hotspot as long as the library is verifiable within the JVM heap boundary and the modularization would be more efficient when an application utilizes multiple instances of libraries.

VII. Related Work

Race detection tools based on static analysis techniques typically sacrifice completeness, in the sense that they can only deal with a particular set of programming idioms, and thus disallow legal data-race free programs. Some tools deliberately sacrifice soundness for scalability, failing to identify certain data races. For example, Chord [16], which can handle lexically-scoped lock-based synchronization, fork/join synchronization, and wait/notify, starts by constructing a superset of possible conflicting operations, then filters this set using a sequence of analyses, and reports a possible data race for all remaining pairs. Another example is the rcc checker [17] as recently resurrected and extended for the Mobius project [18]. This tool uses a type theory base approach (which requires annotations by the users) to ensure that locking is done correctly. In its most recent incarnation, it also recognizes that volatile variables do not need to be protected by locks to avoid data races. However, in whatever form, the tool cannot deal with happens-before edges obtained via transitivity and generates false positives as a result.

Tools that perform dynamic race detection look for races in particular executions of the program. The disadvantage is that dynamic tools only detect problems in the test cases that are actually examined. These are typically based on maintaining vector clocks or the lock-set algorithm with checks to see if every shared variable access is consistently locked. Eraser [19] is an influential example of a lock-set based detector. The tool most closely related to JRF is Goldilocks[20]. Goldilocks is a dynamic analysis tool using an algorithm based on a relation that is very similar to the inverse of h. In other words, the Goldilocks algorithm maintains a function for each variable that indicates which threads can access the variable. As with all tools performing dynamic analysis, the required instrumentation of the program may change its behavior and the tool is limited to analyzing paths that happen to be tested.

Race Free Java is a type system for a simplified version of Java that statically prevents races by allowing the type system to ensure that each object is consistently locked, is immutable, or is local to a single thread. It cannot deal with other widely used concurrent programming idioms such as those using volatile variables, the java.util.concurrent.atomic classes, barriers, detecting termination, etc [21]. Parameterized RaceFreeJava extends RaceFreeJava to information about object ownership [22], [23].

Bounded model checking has been used for the last several years and considers a finite prefix of a path with length k by unrolling the finite state machine for k steps [24]. The verification of the module within the universal environment is an example of bounded model checking.

Bandera automatically generates an environment from environment assumptions provided as LTL formulas or regular expressions [25]. The generated environment is an abstraction that approximates the actual environment whereas in this work the universal environment is general. The Calvin checker [26] is a modular approach using assume-guarantee model checking. It uses user specifications about environment assumptions to constrain thread interactions based on locking. There are no contraints applied on thread interactions in our approach. Environments for components automatically using side-effects and points-to analyses in modular model checking [27]. An interface grammar is used to generate component stubs to use in compositional model checking [28]. Precise component interfaces are generated using learning techniques during state

space traversal [29]. The techniques for interface generation cannot be applied to data race detection since the happens-before ordering between modules cannot be specified through an interface.

Static analysis has been used for environment generation. A dataflow analysis has been used to generate the most general environment of an open reactive system [30]. Static analysis has been used to generate a reasonable behavior model of the *artificial environment* by focusing on parallelism [31]. These techniques, however, cannot detect data races.

VIII. Conclusion

Detecting data races in relaxed memory models is a difficult problem. We present a novel modular verification technique to ensure race freedom in Java programs in the JMM. A library or module developer annotates methods in the module with preconditions that ensure data race freedom. Given the preconditions, we generate a universal environment that invokes methods in the module with symbolic variables. Symbolic execution and explicit state model checking verify whether the preconditions indeed guarantee race freedom. To the best of our knowledge this is the first use of symbolic execution with a universal environment. If additional races are detected in the verification process then the developer strengthens the preconditions or changes the program and repeats the verification process. Finally, the application that uses the module is checked to ensure that it satisfies the module's preconditions and does not introduce its own data races. Future work is to extend the h precondition to more powerful h invariant that captures the transitivity of happens-before relationships required to guarantee the race freedom using lock-free algorithms, and to relax the bounding in the number of threads and depths that limits the applicability of this approach. We have an end to end implementation within the JPF tool kit for the JMM where we extend JRF race detector to perform the modular analysis. We believe, however, that our approach can be applied to other memory models, such as C# and C++.

Acknowledgment

The work of Kyunghee Kim was in part funded by MCT and NASA Ames Research Center. The authors would also like to thank Dr. Tuba Yavuz-Kahveci of University of Florida for her valuable comments.

References

[1] S. Burckhardt, R. Alur, and M. Martin, "Checkfence: checking consistency of concurrent data types on relaxed memory models," in *PLDI*. ACM, 2007, pp. 12–21.

[2] S. Burckhardt and M. Musuvathi, "Effective program verification for relaxed memory models," in *CAV*. Springer, 2008, pp. 107–120.

[3] R. Guerraoui, T. Henzinger, and V. Singh, "Software transactional memory on relaxed memory models," in *CAV*. Springer, 2009, pp. 321–336.

[4] K. Kim, T. Yavuz-Kahveci, and B. A. Sanders, "Precise data race detection in a relaxed memory model using heuristic-based model checking," in *ASE*, 2009.

[5] ——, "JRF-E: Using model checking to give advice on eliminating memory model-related bugs," in *ASE*, 2010.

[6] J. Manson, W. Pugh, and S. V. Adve, "The Java memory model," in *POPL*. ACM Press, 2005, pp. 378–391.

[7] D. Aspinall and J. Sevcik, "Formalising Java's data-race-free guarantee," in *TPHOLs 2007 (LNCS)*, vol. 4732. Springer, 2007, pp. 22–37. [Online]. Available: http://groups.inf.ed.ac.uk/request/jmmform.pdf

[8] T. Xie, D. Marinov, W. Schulte, and D. Notkin, "Symstra: A framework for generating object-oriented unit tests using symbolic execution," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 365–381, 2005.

[9] S. Khurshid, C. Pasareanu, and W. Visser, "Generalized symbolic execution for model checking and testing," *TACAS*, pp. 553–568, 2003.

[10] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda, "Model checking programs." *Automated Software Engineering*, vol. 10, no. 2, pp. 203–232, 2003.

[11] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing NASA software," in *ISSTA*, 2008, pp. 15–25.

[12] C. Păsăreanu and N. Rungta, "Symbolic PathFinder: symbolic execution of Java bytecode," in *ASE*, 2010, pp. 179–180.

[13] K. Kim, T. Yavuz-Kahveci, and B. A. Sanders, "Precise data race detection in a relaxed memory model using model checking," University of Florida, Tech. Rep. REP-2009-480, 2009.

[14] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[15] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, *Java Concurrency in Practice*. Addison Wesley Professional, 2006.

[16] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for Java," in *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM Press, 2006, pp. 308–319.

[17] C. Flanagan and S. N. Freund, "Type-based race detection for Java," in *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM Press, 2000, pp. 219–232.

[18] "Mobius consortium. deliverable d3.3: Preliminary report on thread-modular verification," March 2007, http://mobius.inria.fr.

[19] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, 1997.

[20] T. Elmas, S. Qadeer, and S. Tasiran, "Goldilocks: a race and transaction-aware Java runtime," in *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2007, pp. 245–255.

[21] M. Abadi, C. Flanagan, and S. N. Freund, "Types for safe locking: Static race detection for Java," *ACM Trans. Program. Lang. Syst.*, vol. 28, no. 2, pp. 207–255, 2006.

[22] C. Boyapati, R. Lee, and M. Rinard, "Ownership types for safe programming: Preventing data races and deadlocks," in *OOPSLA*. ACM, 2002, pp. 211–230.

[23] R. Agarwal and S. D. Stoller, "Type inference for parameterized race-free Java," in *VMCAI*. Springer-Verlag, 2004, pp. 149–160.

[24] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in computers*, vol. 58, pp. 117–148, 2003.

[25] O. Tkachuk and M. B. Dwyer, "Automated environment generation for software model checking," in *ASE*, 2003, pp. 116–129.

[26] C. Flanagan, S. Qadeer, and S. A. Seshia, "A modular checker for multithreaded programs," in *CAV*. Springer, 2002, pp. 180–194.

[27] O. Tkachuk and M. B. Dwyer, "Adapting side effects analysis for modular program model checking," *SIGSOFT Softw. Eng. Notes*, vol. 28, no. 5, pp. 188–197, 2003.

[28] G. Hughes and T. Bultan, "Interface grammars for modular software model checking," in *ISSTA*, 2007, pp. 39–49.

[29] D. Giannakopoulou and C. S. Păsăreanu, "Interface generation and compositional verification in JavaPathfinder," in *FASE*. Springer-Verlag, 2009, pp. 94–108.

[30] C. C. Loyola, C. Colby, P. Godefroid, and L. J. Jagadeesan, "Automatically closing open reactive programs," in *PLDI*. ACM Press, 1998, pp. 345–357.

[31] P. Parizek, J. Adamek, and T. Kalibera, "Automated construction of reasonable environment for Java components," *El. Not. Th. Com. Sci.*, vol. 253, no. 1, pp. 145–160, 2009.